MC/DC COVERAGE FOR REQUIREMENTS SPECIFICATIONS

by

Gourav Das

April, 2018

Director of Thesis: Sergiy Vilkomir, PhD

Major Department: Computer Science

In the early 1990s, the Modified Condition/Decision Coverage (MC/DC) criterion was suggested as a structural white-box testing approach, but it can also be used for black-box specification-based testing. Practical application of MC/DC for specification-based testing has its own unique features and is sometimes quite different from code-based applications. However, MC/DC as a black-box approach has not been studied sufficiently, and thus, the application of MC/DC for specification coverage was the main research problem considered in this thesis. The goal of this study was to analyze MC/DC as a black-box technique, investigate factors that distinguish black- and white-box applications of this approach, and provide proper definitions and rules with a prototype implementation to evaluate the MC/DC level during black-box testing.

MC/DC COVERAGE FOR REQUIREMENTS SPECIFICATIONS

A Thesis

Presented to The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment of the Requirements for the Degree

Master of Science in Software Engineering

by

Gourav Das

April, 2018

Copyright Gourav Das, 2018

MC/DC COVERAGE FOR REQUIREMENTS SPECIFICATIONS

by

Gourav Das

APPROVED BY:

DIRECTOR OF THESIS: _____

Sergiy Vilkomir, PhD

COMMITTEE MEMBER: _____

Mark Hills, PhD

COMMITTEE MEMBER: _____

M.N.H Tabrizi, PhD

CHAIR OF THE DEPARTMENT

OF COMPUTER SCIENCE: _____

Venkat Gudivada, PhD

DEAN OF THE

GRADUATE SCHOOL: _____

Paul J. Gemperline, PhD

ACKNOWLEDGEMENTS

First, I would like to thank my thesis advisor Dr. Sergiy Vilkomir of the department of Computer Science at East Carolina University. Thank you for all the useful comments, remarks and engagement through the learning process of this master thesis. While applying for my Masters program in the United States, Dr. Vilkomir was my first preference because he is one of the few professors conducting research on software testing and quality assurance. The door to Dr. Vilkomirs office was always open whenever I had a question about my research or writing. He consistently allowed this thesis to be my own work, but steered me in the right direction whenever he thought I needed it. Every time, I left his room after a discussion I felt incredibly motivated towards my research work. This thesis would not have been possible without his guidance and supervision.

I would also like to thank all the representatives of froglogic GmbH for allowing me to test their software Squish Coco tool and especially I would like to thank Roland Baer from Verifysoft Technology GmbH for his extensive role in explaining the installation process and the basic functionality of the CTC++ tool.

Table of Contents

LIST OF FIGURES

Chapter 1

DEFINITIONS, ACRONYMNS, ABBREVIATIONS, AND SYMBOLS

Boolean Expression: This is an expression which evaluates either as a False or True outcome. Boolean Expressions are also abbreviated as F/0 for False and T/1 for True [1].

Boolean Function: A function that returns a Boolean value (False, True). It may be either user defined or implementation defined. The relational operators operating on non-Booleans are examples of implementation defined Boolean functions [1].

Condition: This is a leaf-level Boolean expression which cannot be broken down further into simpler Boolean expressions.

Decision: This is a Boolean expression which controls the flow through a program, for example, if or while statement. Decisions may be composed of a single condition or a combination of multiple conditions.

Statement Coverage: This verifies whether or not every statement in the program has been executed at least once.

Decision Coverage: This verifies whether or not every decision within a program has considered all possible outcomes at least once.

Condition/Decision Coverage: Every condition within a decision of a program has considered all possible outcomes at least once, and all possible outcomes are considered at least once. MC/DC (Modified Condition/Decision Coverage): In a program

where every point of entry and exit has been invoked at least once, every condition in a decision has taken on all possible outcomes at least once, and it is demonstrated that every condition affects that decision outcome independently. Coupled Conditions: It is a situation where two or more conditions are involved, where changing one condition causes change in the other conditions [1].

Strongly coupled condition: It is a situation where changes in one condition always changes the others. For an instance, in the expression

(X = 0 and A) or (X 0 and B)

Here, it is clearly visible that both X=0 and X0 conditions are strongly coupled. So, changing the value of X always changes both conditions [1].

Weakly coupled conditions: It is a situation where changing one condition sometimes changes the other conditions.

For an instance, in the expression

X = 0 or X = 1 or X = 3

Here, X = 0, X = 1 and X=3 conditions are weakly coupled. When we change the value of X from 0 to 2, it only changes the first condition, whereas when we change the value of X from 0 to 1, then the first two conditions also change [1].

LHS: It stands for Left-Hand Side. LHS is an operand which stays on the left-hand side of a binary infix operator [1].

RHS: It stands for Right-Hand Side. It is an operand which stays on the right-hand side of a unary/binary infix operator [1].

Masking: A process where setting the RHS/LHS operand of an operator to a value so that when we change the LHS/RHS operand of that operator then it does not change the value of the operator [1]. For instance, in case of an AND operator, the following two expressions will always give a false result: X AND False = False AND X = False, no matter what is the value of X is. Similarly, in case of an OR operator,

the following two expressions will always give a true result: X OR True = True OR X = True, no matter what is the value of X is [1].

Masking MCDC: A form of MCDC where a conditions independence could be shown by allowing all possible forms of masking [1]. **Short-Circuit Form Boolean Operator:** A binary Boolean operator where the LHS is evaluated first, and then the RHS is evaluated conditionally. For instance, when we use the short-circuit AND, if the LHS is False, then False is returned as the result and the RHS is not evaluated. If the LHS is True, then the RHS is evaluated and that result is returned [1].

SRS: Software Requirements Specifications.

JDBC: Java Database Connectivity.

Chapter 2

Introduction

In the early 1990s, the need for having a better code coverage was realized to ensure that no condition in a program is present for no reason. That was how Modified Condition/Decision Coverage (MC/DC) criterion was suggested as a structural white-box testing approach and became required by the DO-178B Standard [2] for testing avionics software. The goal of MC/DC is this: thorough testing of logical expressions (predicates) for safety-critical software, to ensure that each condition in the code has been tested to independently effect the decision.

This coverage is needed to ensure that all the conditions written in the code are present for a specific, desired purpose. This is extremely important, because if we fail to understand the purpose of even one condition, then that could end in a disaster where human life or a huge financial interest is in danger. To ensure such risk are minimized, the concept of MC/DC came into existence.

For safety-critical software, the complexity is so high that often times, organizations like NASA or SpaceX will use redundancy to minimize the probability of failure. However, instead of focusing much on redundancy we would rather focus on improving the way MC/DC has been calculated by several tools in the market and also we will focus on how we could define MC/DC coverage to reduce the probability of the failure by defining it for the Requirements Specifications.

While MC/DC was developed as a white-box (code-based) criterion, it was clear from the beginning that it could be used "to guide the selection of test cases at all levels of specification" [3]. Using MC/DC for black-box specification-based testing is especially important for safety-critical systems when a decision can be associated with a systems critical operation. For example, a decision may be responsible for actuating a reactor protection system at a nuclear power plant in which the conditions in this decision describe various criteria for actuation, e.g., high temperature or low pressure [4].

Practical application of MC/DC for specifications coverage has its own unique features and sometimes differs significantly from code-based applications. Even the rules of the MC/DC level evaluation can vary in these two cases. However, MC/DC as a black-box approach has not been studied sufficiently, and most research on MC/DC has treated it as a white-box approach [5].

The goal of this thesis was to analyze MC/DC as a black-box technique, investigate factors that distinguish the black- from the white-box applications of this approach, provide proper definitions and rules to evaluate the MC/DC level during black-box testing and develop a tool to practically implement a prototype for the definition of MC/DC. Using MC/DC as a black-box specification-based testing method not only involves test generation from the logical specifications, but also the estimation of the MC/DC coverage level of formal logical specifications (not a code) achieved for an arbitrary set of test cases. Thus, the application of MC/DC for specification coverage was the main research problem considered in this thesis.

The thesis is organized as follows: Chapter 2 provides a contrastive analysis of MC/DC from the white- and black-box approach. Specific features and factors that differ between these two approaches were determined and investigated. Chapter 3 analyzes approaches used to estimate the MC/DC level for specification-based testing

and the potential to apply existing coverage tools for this purpose. This section also provides background and experience with the various tools used for this purpose. Chapter 4 provides information on implementation details of the tool where the Agile Principle has been used to minimize documentation and keep the entire development process simple to enhance agility. This portion describes the entire requirements for the implementation using User Stories which were used instead of writing a very large SRS documentation. To describe the entire implementation process, scrum software development cycle has been utilized, where each of the user stories has been shown to transform through product backlog, sprint backlog into smaller tasks, which finally delivers a shippable product. Chapter 4 also includes details of tool interface, Use Case, Class Diagram, code details, and the backend SQLite database, which has been used to develop the tool. Chapter 5 provides a brief review of related work, and the section of the thesis document presents the conclusions and directions for future work.

Our main contribution and novelty are in two areas:

1. Investigation of not completed (not 100%) MC/DC coverage:

- We considered a problem of MC/DC evaluation when test cases do not provide completed 100% MC/DC coverage. This is a new research direction, important both from theoretical and practical point of view.
- This problem has not been investigated in the research literature. According to our knowledge, our paper is the first one devoted to this problem.
- To investigate practical applications, we used three different testing coverage tools and analyze, why these tools provide different results for the same test sets. The list of factors important for practical coverage evaluation is provided. These results would be useful for testing of safety-critical software.
- We suggested a simple definition and approach to not completed MC/DC evaluation and implemented it in a prototype of testing coverage tool.

2. Application of MC/DC for specification coverage (black-box MC/DC):

- This problem has not been studied enough in research literature; the majority of research focuses on traditional code-based (white-box) MC/DC.
- We have shown the importance of black-box MC/DC, especially for safety-critical software.
- Our results proved that many problems of code coverage are not significant for specification coverage and can be easily avoided.
- We applied our prototype tool to specification coverage in several cases and achieved actual MC/DC measures, which were corresponded to manually calculated expected results.
- We hope that, after future development, this tool can be successfully used in practical software testing.

The initial work on this thesis, which includes the main results, have been presented and published in the Proceedings of the 28th Annual IEEE Software Technology Conference (STC 2017), September 25-28, 2017, Gaithersburg, MD, USA [6].

Chapter 3

CONTRASTIVE ANALYSIS OF MC/DC FROM THE WHITE-BOX AND BLACK-BOX PROSPECTIVES

White-box and black-box testing are well-established terms in software testing for many years [7]. Thus, according to IEEE Std 610.12-1990 "IEEE Standard Glossary of Software Engineering Terminology," white-box testing is the same as structural testing and is defined as "Testing that takes into account the internal mechanism of a system or component" [8]. In contrast, black-box testing is the same as functional testing and is defined as "Testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions" [8]. In other words, white-box is code-based testing, and black-box is specification-based testing.

For coverage criteria, this terminology should be specified. Such criteria, including MC/DC, can be used for several purposes [9]:

- as stopping rules to decide whether sufficient testing has been done
- as generators to generate test cases according to some criterion, or
- as measurements to evaluate test quality based on coverage percentage.

Therefore, test cases can be generated based on specifications (black-box) and then be evaluated based on code coverage (white-box) according to some criterion.

The same criterion often can be used for evaluation of test case quality based on specifications coverage (black-box).

In this thesis, we do not consider test case generation and solely investigate the task of coverage evaluation according to MC/DC as a test quality measurement. The term white-box MC/DC is used when MC/DC is applied for code coverage evaluation, and black-box MC/DC is used when MC/DC is applied for specifications coverage evaluation.

The initial definition of MC/DC is as follows: "Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to independently affect the decisions outcome. A condition is shown to independently affect a decisions outcome by varying just that condition while holding fixed all other possible conditions" [2] [3]. The beginning of the definition is not relevant to black-box testing, but the principal idea (independent effect on a decision) can be applied in both white- and black-box approaches.

3.1 White- vs. black-box MC/DC

While the MC/DC definition appears to be quite clear, many questions arise in different situations that require further clarification before the practical application of MC/DC is possible. Sometimes, these questions provide different answers and lead to various forms of MC/DC such as weak and strong MC/DC [3] or unique-cause, unique-cause + masking, and masking MC/DC [9]. We selected the following factors that are important in MC/DC application and are the sources of the differences between white- and black-box MC/DC:

- object of coverage (correct specifications vs. possible faulty implementation)

- logic representation (Boolean function vs. Boolean expression)
- short-circuit logic
- masking problems
- multiple occurrences of conditions
- language- and compiler-sensitivity of logical expressions
- dependencies among conditions (condition coupling)
- dependencies among decisions.

In many cases, black-box MC/DC is simpler and more intuitive than white-box MC/DC. We demonstrate this with the example of the logical specification A∧B, where A and B are logical variables with possible values True (T) or False (F). Despite its simplicity, such specification is quite typical for many formal specifications of software systems. Thus, for the Flight Guidance System, it describes when to turn off the Flight Director (FD) [1]. Here A means that the FD switch is pressed, and B means that there is no over speed condition (Fig. 3.1). For a nuclear power plant safety system, such logical expression can describe some specific safety property. For example, A means that a water level in the boiler is at least 3 meters, and B means that the reactor temperature is less than 3,000 degrees [10].

When_Turn_FD_Off

Condition:

AND	When_FD_Switch_Pressed_Seen$_{\text{m-103}}$()	T
	Overspeed_Condition$_{\text{m-128}}$()	F

Figure 3.1: Example of the Flight Guidance System specification

Assume that three test cases are used for testing the considered A∧B specification: TT (means A=T and B=T), TF, and FT. Evaluating the quality of these test cases

based on MC/DC specification coverage, it is easy to see that the MC/DC level equals 100%. Indeed, A is covered by TT and FT and B is covered by TT and TF, so the quality is good from this point of view. These test cases are also reasonable from the practical point of view because they test the situations when the system should fulfill some important actionturn off a piece of equipment, provide a safety signal, etc.

Assume now that the quality of these test cases is evaluated based on MC/DC code coverage. The implementation (code) could be unknowingly faulty and, for example, by mistake contain A∨B instead of A∧B. The MC/DC evaluation of this faulty code shows that neither A nor B is covered, so the quality of the test cases is not good. This confusing conclusion contradicts the specification evaluation and requires unnecessary action for test improvement.

The same example of the A∧B specification can demonstrate differences between white- and black-box MC/DC for masking, short-circuit logic, and other language- and compiler-sensitivity issues. When A=F, the whole expression equals F whatever the value of B is. In many cases, software evaluates A and the expression, but does not assign any value to B. Thus, two questions arise:

- Because the value of B is not assigned during execution when A=F, should A be considered covered by TT and FT?
- Because the value of B does not affect the value of the expression when A=F, should A be considered covered by TT and FF?

For code-based evaluation, different answers are considered in practice. For black-box MC/DC, the answers are more straightforward. The goal of testing is to consider situations with specific values of A and B and to check that software provides correct outputs for these input values. Testing ignores the details of the implementation, so the most natural answers are “Yes” for the first question and “No” for the second question.

Another factor is the problem of multiple occurrences of conditions in a decision. For example, should we count condition A in the decision $(A \wedge B) \vee (\neg A \wedge C)$ as one condition or two? For white-box MC/DC, both possible answers were considered: to treat A as a single condition (weak MC/DC) and to view each condition as a distinct entity (strong MC/DC) [3]. The second option is slightly artificial and makes the situation confusing. For black-box MC/DC, this logical specification can be considered as a Boolean function with arguments A, B, and C, so A is naturally counted only one time.

A similar problem arises when two logical expressions look different but are equivalent and have the same truth table. For white-box MC/DC, a test set can demonstrate different levels of MC/DC for these expressions. For black-box MC/DC, only the expected software behavior is important, not the form of logical expressions inside the code. The black-box MC/DC level can be evaluated directly from the truth table, so a test set can provide the same MC/DC values for both expressions.

Chapter 4

Estimation of an MC/DC Level for Black-Box Specification-Based Testing

The reasons for the existing different approaches to MC/DC estimation are the different types of MC/DC and the variety of practical situations during MC/DC application. While definitions for 100% MC/DC are well known and commonly accepted, definitions for situations with less than 100% MC/DC and common approaches to MC/DC estimation are absent.

For black-box MC/DC, we suggest that the most natural approach is to estimate each condition separately. Each condition can be covered completely (100% MC/DC) if there is a pair of test cases according to the MC/DC requirements, and coverage can be absent (0% MC/DC) if such a pair does not exist. The MC/DC coverage of a decision is then the percentage of the number of conditions covered compared to the total number of conditions in a decision.

It is not possible to generate test cases or evaluate MC/DC manually for large-scale software, so tools should be used for these purposes in practice. Most MC/DC coverage tools measure the coverage of code, and only a few tools measure the MC/DC of logical requirements/specifications. These tools have a narrow area of application and work with specific models, specification language, or other formal environments.

Thus, for Simulink models, which are represented graphically as block diagrams,

Simulink Verification and Validation [11] and Simulink Design Verifier [12] tools by MathWorks provide a measurement of coverage. These tools produce several coverage metrics including MC/DC. Different MC/DC definitions are used depending on the applied settings (see more details in [13], [14]).

One would think that the simple solution is to enter logical specifications (just a list of logical expressions without corresponding functionality) in a simple software program and then use code-coverage tools. However, this approach does not work properly in practice. The problem is that different tools often provide different results even when 100% is expected. For less than 100%, MC/DC results are nearly always different, and there is no common opinion about which results should be considered correct. We illustrate this situation in Figure 4.1 and Figure 4.4. Both tables contain the same four logical expressions and the same sets of test cases. The results of MC/DC levels are presented in four columns:

- estimated manually according to our definition above,
- estimated by the Coco tool [15],
- estimated by the CodeCover tool [16], and
- estimated by the Testwell CTC++ tool [17].

4.1 Different approaches to MC/DC estimation

Figure 4.1 contains results for 100% MC/DC levels according to our definition, and Figure 4.4 considers situations with less than 100% MC/DC. Because the Coco and CTC++ tools evaluate the MC/DC coverage for a whole software program (all decisions in the program together), we recalculated the contributions for each separate decision.

Expression	Test set	MC/DC Level, %			
		Black-box, Manual	Coco Tool	Code Cover Tool	Testwell CTC++ Tool
A ∨ B	TF, FT, FF.	100	100	100	100
A ∧ B ∧ C	TTT, TTF, TFT, FTT.	100	100	100	100
A ∨ (B ∧ C)	TFT, FTT, FFT, FTF.	100	100	100	100
	Case 2 TTF, TFT, FTF, TFF.	33	50	50	60
(A ∧ B) ∨ (A ∧ C) ∨ (B ∧ C)	TFT, FTT, FFT, TFF.	100	42.9	75	50
	TTF, FTT, FTF, TFF.	100	57.1	83.3	62.5

Figure 4.1: MC/DC Levels for Generated Test Sets

As Figure 4.1 show, even for 100% MC/DC, the results are only the same for very simple specifications such as A∨B or, for example, A∧B∧C. For less than 100% MC/DC and small numbers of test cases, the results differ dramatically (Figure 4.4). We explain these differences for four cases from the tables.

Before moving ahead lets look into the background of each of the tools used in the MC/DC coverage evaluation.

4.2 Background and experience with the Tools used in the analysis.

CodeCover:-

CodeCover tool [16] was developed by University of Stuttgart as a free white-box testing tool in the year 2007. It measures different types of white-box coverages such as statement, branch, loop, and term coverages (MC/DC coverage). It works on Linux, Windows, Mac OS command line platform, and Eclipse IDE. Java and COBOL languages are supported by this tool.

We installed CodeCover tool and added CodeCover as a plugin in Eclipse IDE. Then we used our code, written in Java language, to test each of the decisions mentioned in the tables in the forthcoming sections. This installation was one of the easiest and most straightforward among all the tools which we have analyzed for this experiment.

Squish coco:-

Squish Coco is a code coverage tool [15] developed by a private company, froglogic, which was founded in the year 2003. It uses Tcl, QML, C# and C/C++ programming languages. It supports Mac OS, Linux and MS Windows operating systems. Squish Coco analyzes the way a code/application runs and produces results which can be used to make tests more efficient and complete.

The main functions of the Squish Coco tool include:

- Identify untested code section
- Identify redundant tests which can then be eliminated
- Identify dead code by displaying the code that was never executed
- Analyze two separate versions of an application and compares the result

The main components of Squish Coco, which have been used for this research, include:

- CoverageScanner, which is a program used to analyze and instrument the application under test.
- CoverageBrowser, which is a program used to display the results of the analyzed coverage.

Running this tool was one of the major challenges of the entire research project, as the installation manual wasnt very user-friendly at the time when we were considering this tool for our research analysis. However, at the end, we have managed to make it work and performed out tests on this tool.

Testwell ctc++:-

Testwell CTC++ is one of the leading code coverage tools offered by Verifysoft Technology [17]. It supports various programming languages like C, C++, Java, and C#. The goal of this tool is to ensure that all parts of the complex code have been tested before release, and also identifies which areas of an application were exercised during a test run. It also integrates easily with your existing build and test infrastructure through a full command line interface.

With Testwell CTC++, we have faced a lot of trouble with command line interface. However, the representative of this company graciously assisted us for several days through TeamViewer application and continuously modified the license file to ensure that the tool runs properly in our systems.

4.3 CASES CONSIDERED FOR THIS EXPERIMENT

4.3.1 CASE 1: EXPRESSION A ∨ B; TEST SET TF, FF FROM Figure 4.4

The test set contains two test cases that cover condition A but not condition B. Therefore, according to our definition, the MC/DC level is 50%.

a	b	Decision	Description
FALSE	TRUE	TRUE	*never executed*
FALSE	FALSE	FALSE	*executed 1 time by 1 test*
TRUE		TRUE	*executed 1 time by 1 test*

Figure 4.2: Coco screenshot for Case 1.

At the same time, the result from the Coco tool is different. This is because only A is covered, and two test cases (marked green in Figure. 4.2) of the three are required for complete MC/DC in this coverage. Therefore according to Coco, the total coverage is (2/3)*100% = 66.7%

Let‘s try to dig in further and see how exactly this tool is calculating MC/DC:

Once we create the MC/DC table (shown in Figure. 4.3) for this Case I, we get a total of four combinations of test cases (TT, TF, FT, FF). Here we can clearly see only 3 of the 4 test cases shows MC/DC. If you notice in Figure. 4.2, only these 3 rows are considered by Squish Coco tool for calculating MC/DC. So, it is obvious when you insert TF, FT, FF as three test cases in this tool you get 100% MC/DC.

In the last row of Figure. 4.2, the value for condition B have been masked. That means both the possible values of condition B would result in MC/DC coverage, which is strictly violating the rules of MC/DC coverage.

A	B	Decision	A*	B*	MC/DC
T	T	T			
T	F	T	*		*
F	T	T		*	*
F	F	F	*	*	*

Figure 4.3: MC/DC coverage for Case 1

Figure. 4.3 is the MC/DC coverage for Case 1, we can take this table into consideration to understand what are the different approaches by which we can calculate MC/DC coverage in general. At this stage, we know what conditions are giving test MC/DC coverage which is both A and B. But the question is what is the best approach to calculate MC/DC coverage for this particular table.

To answer this, we suggest two important levels of factors to determine MC/DC coverage:

- Higher Level: We have to first determine at a higher level whether we want to calculate MC/DC coverage based on the column approach or the row approach.
- Lower Level: Then on a lower level we have to determine whether we want to consider various factors which distinguish between black-box MC/DC and white-box MC/DC like masking, short circuit, repetition of conditions, compiler and language sensitivity. Also we must consider some other type of factors which are not specific to white-box MC/DC or black-box MC/DC but are more general like whether to cover all possible decisions in the MC/DC coverage separately as a coverage criterion, and shall we consider a condition for calculating MC/DC which shows half MC/DC coverage.

In higher level factor for calculating MC/DC, column-wise approach (our definition of MC/DC) is based on the core definition of MC/DC where we evaluate each condition independently to see if each condition is individually impacting the deci-

sion or not. So, it is very important that the calculation of MC/DC should be purely based on just the condition itself and not based on the combination of different conditions which is the row approach (definition of MC/DC in various tools like Squish Coco and CodeCover). In Figure 4.2, you can see the MC/DC coverage criteria have been calculated based on a combination of conditions (A and B) which is not what we define in the actual definition of MC/DC. We never define MC/DC coverage as a combination of different conditions which shows MC/DC.

When we calculate MC/DC for more complex equations where there may be a huge number of conditions, there it is highly likely that the test manager will have a hard time to realize if the tool is giving accurate MC/DC coverage or not. For instance, if there are 20 conditions in an equation then you need to make a table of 20 conditions MC/DC or have a solid idea of how Squish Coco is creating MC/DC for your 20 conditions. This is always complex. But if you rather follow our manual approach, you can easily find how many conditions of 20 are not fulfilling MC/DC coverage. Simply calculate MC/DC for each condition individually. So, using our approach, if you get 95% coverage that means 19 of 20 conditions are fulfilling MC/DC coverage. For a user, this is always an easier approach to calculate and analyze the result of MC/DC coverage.

Expression	Test set		MC/DC Level, %		
		Black-box, Manual	Coco Tool	Code Cover Tool	Testwell CTC++ Tool
A ∨ B	TT	0	0	25	25
	FT	0	0	25	25
	Case 1 TF, FF.	50	66.7	75	75
	FT, FF.	50	66.7	75	75
A ∧ B ∧ C	TTT	0	0	50	20
	FFF	0	0	16.7	20
	TTT, FTT.	33.3	50	66.7	60
	TTT, TFT.	33.3	50	66.7	60
	TTT, FTT, TFT.	66.7	75	83.3	80
	TTT, TFT, TTF.	66.7	75	83.3	80
A ∨ (B ∧ C)	TFF	0	0	16.7	20
	FTF	0	0	33.3	20
	TFT, FFT.	33.3	50	50	60
	TTF, TFF.	33.3	0	16.7	20
	Case 3 TFT, FFT, FTT.	66.7	75	83.3	80
	TTF, TFF, TFT.	66.7	0	16.7	20
(A ∧ B) ∨ (A ∧ C) ∨ (B ∧ C)	Case 4 TTT	0	0	16.7	12.5
	FFF	0	0	25.0	12.5
	TTF, FTF.	33.3	28.6	41.7	37.5
	FTT, FTF.	33.3	28.6	41.7	37.5
	TTF, FTF, TFF.	66.7	42.9	66.7	50
	FTT, FTF, TTF.	66.7	42.9	58.3	50

Figure 4.4: MC/DC Levels for Incomplete Test Sets

The CodeCover tool uses a different approach and calculates the contribution to the coverage separately for each test case. The first test case provides 50% total coverage (i.e., 50% for both A and B). The second test case provides 25% coverage

(50%, but only for A). Therefore, the total coverage is 50% + 25% = 75% (Fig. 3).

When you dig into how this tool is calculating MC/DC coverage for Case I, you will get to see for condition B (marked in green in Figure 4.5), in the first test case, F has been considered to cover half of the MC/DC because the value of condition B in the second test case has been masked. There is one more reason why F has been considered for MC/DC because both values of F and the result is same. Otherwise, this tool wouldnt consider F in condition B for MC/DC. This shows the developer of this tool has given benefit of doubt to condition value of B in test case one and hence it is giving 50% coverage. But MC/DC shouldn''t be calculated based on probability. Each condition should either give 100% MC/DC coverage or zero percent MC/DC coverage. There should be nothing in between. That is why this approach of calculating MC/DC coverage in this tool is not appropriate.

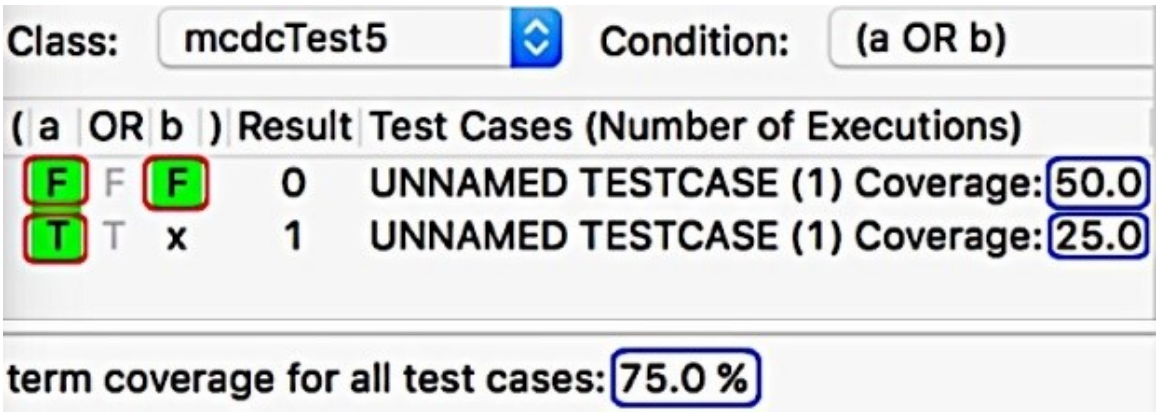

Figure 4.5: CodeCover screenshot for Case 1

The result from CTC++ is the same as from CodeCover; however, this tool uses a different approach and calculates the covered points for each decision: one point when a decision equals T, one point when a decision equals F, and one point for each independently covered condition (Figure. 4.6). Therefore, in this case, a total of four points is possible but only three points are covered (two points for the decision and

one for condition A). The MC/DC is evaluated as (covered points)/(total expression points), which yields (3/4)*100% = 75%.

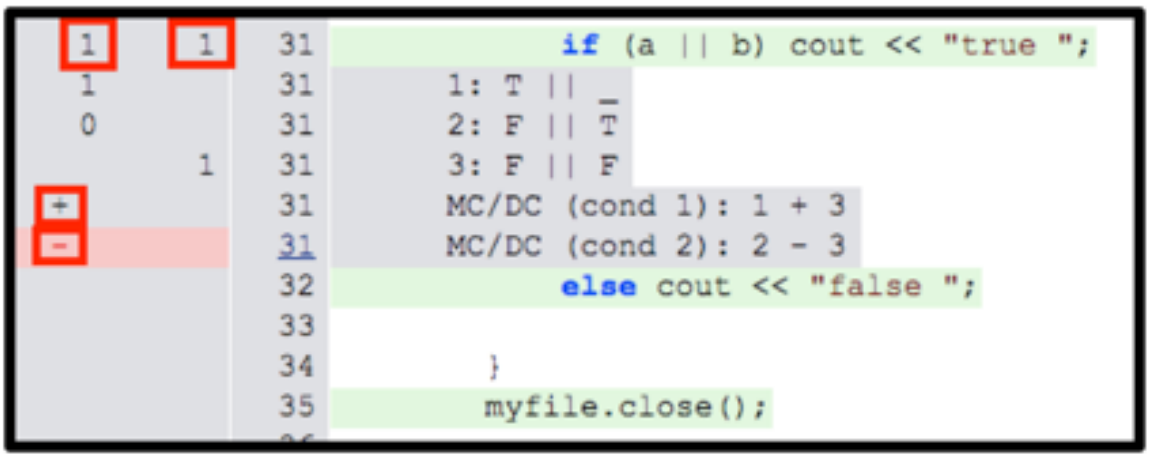

Figure 4.6: Testwell CTC++ screenshot for Case 1

If you dig into how the CTC++ tool is calculating MC/DC coverage you will see (in Figure 4.6) that it uses the same exact approach which we have suggested for calculating MC/DC coverage: by calculating coverage for each condition separately. The only difference is that it also considers both the possible values of the decision as a coverage criterion. However, all the possible values of decisions should never be considered separately because each of the possible values of a decision is always, by default, covered when you calculate MC/DC for each condition independently. For an instance, consider the following Figure 4.7, case I, when we say condition 1 is fulfilling MC/DC, we mean that for all possible values of condition 1 (TF), there is always all possible values covered for a decision too, as shown in Figure 4.10. That is why we should not consider all possible values of decision separately as it is always covered as a subset of the MC/DC coverage for each condition.

A	B	Decision	A*	B*	MC/DC
T	F	T	*	N/A	*
F	F	F	*	N/A	*

Figure 4.7: MC/DC coverage for condition 1 in Case 1

4.3.2 Case 2: Expression A∨(B∧C); test set TTF, TFT, FTF, TFF from Figure 4.1

The test set contains four test cases that provide 33% MC/DC coverage according to our definition of black-box MC/DC: TTF and FTF cover A; B and C are not covered. However, both Coco and CodeCover provide 50% coverage. The Coco tool considers A alone as a covered condition. B and C are not considered covered because their values are masked. Two test cases of the four required for complete MC/DC contribute to coverage for A, which results in 50% (Figure 4.8).

a	b	c	Decision	Description
FALSE	TRUE	TRUE	TRUE	*never executed*
FALSE	TRUE	FALSE	FALSE	*executed 1 time by 1 test*
TRUE			TRUE	*executed 3 times by 1 test*
FALSE	FALSE		FALSE	*never executed*

Figure 4.8: Coco screenshot for Case 2

CodeCover also provides 50% MC/DC coverage but uses a different algorithm with separate contributions from each test case (Figure 4.9). Also, it is not considering value T from condition b as MC/DC because the decision value is a zero which is

false. According to the MC/DC rule, the condition value and decision value should be the same for a particular test case. This is one of the mandatory criteria which we have even used in our code to verify if the test case is showing MC/DC or not. In Figure 4.9, the first row, have one test case line where we are getting 2/3 MC/DC conditions covered and in the second row we are getting only 1/3 condition covered for 3 different test cases starting with True value. All those test cases in the second row have been considered as one because of masking so all the test cases which is starting with True will be shown as one single row by CodeCover tool (Figure 4.9). Overall, in this case, this tool are considering 6 (3+3) conditions out of which 3 (1+2) is covered that is why it is showing 50% MC/DC coverage in Figure 4.9.

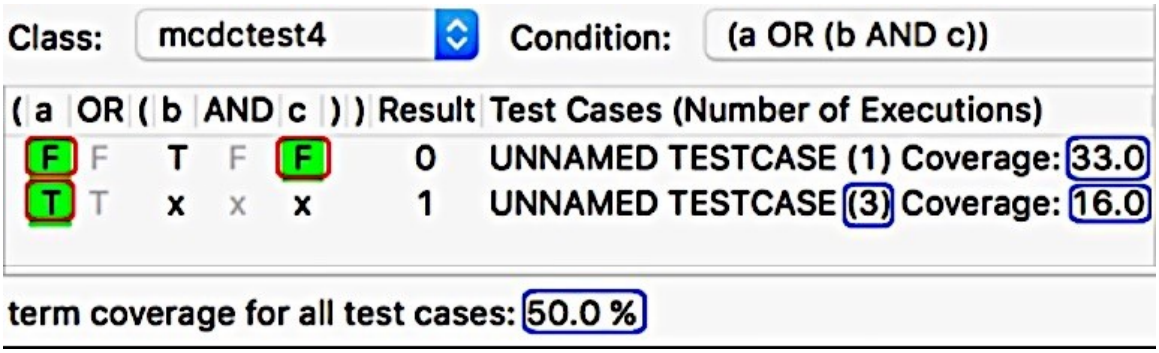

Figure 4.9: CodeCover screenshot for Case 2

Compared with the other tools, the CTC++ tool gives a different result for this case and calculates MC/DC the following way (Figure 4.10):

- two points for the decision
- one point for the covered condition A (TTF and FTF)
- no points for conditions B and C because they are not covered

Altogether, three out of five possible points yields (3/5)*100% = 60%.

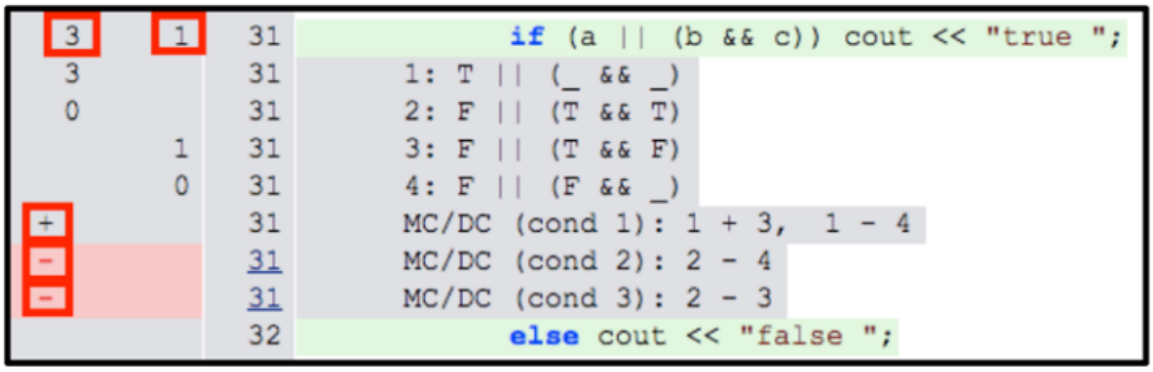

Figure 4.10: Testwell CTC++ screenshot for Case 2

4.3.3 Case 3: Expression A ∨ (B ∧ C); test set {TFT, FFT, FTT} from Figure 4.4

We estimated MC/DC manually from these three test cases as 66.6% because they cover A (TFT and FFT) and B (FFT and FTT), but do not cover C. However, the tools provide 75%, 83.3%, and 80% coverage, respectively (Figures 4.11, 4.12, and 4.13)

a	b	c	Decision	Description
FALSE	TRUE	FALSE	FALSE	*never executed*
FALSE	TRUE	TRUE	TRUE	*executed 1 time by 1 test*
FALSE	FALSE		FALSE	*executed 1 time by 1 test*
TRUE			TRUE	*executed 1 time by 1 test*

Figure 4.11: Coco screenshot for Case 3

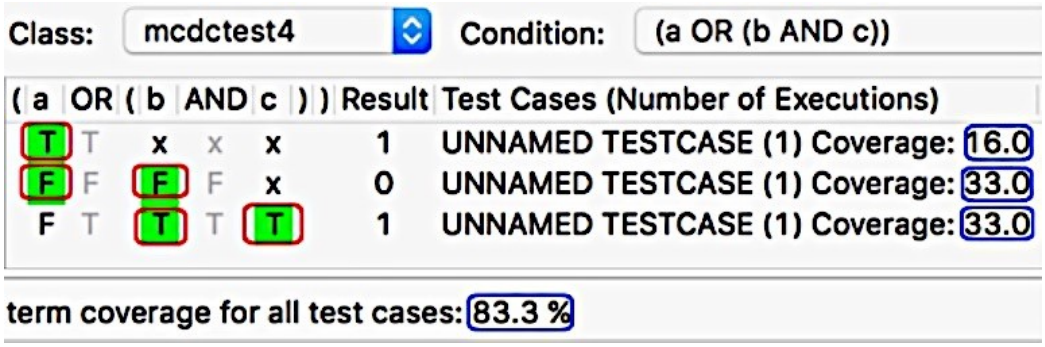

Figure 4.12: CodeCover screenshot for Case 3

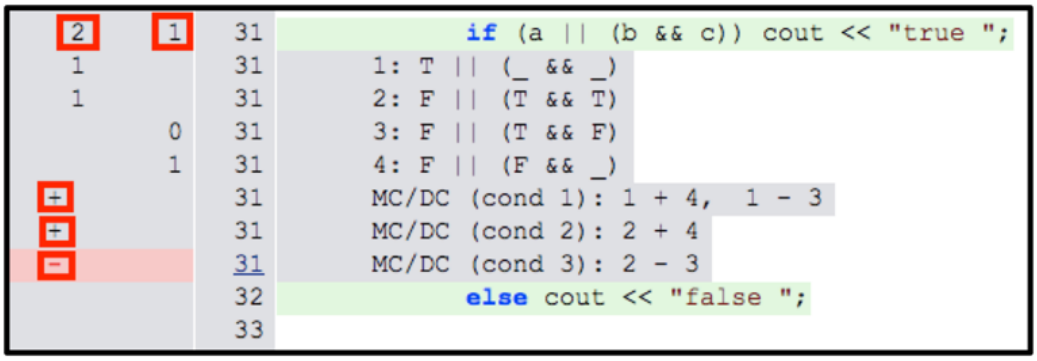

Figure 4.13: Testwell CTC++ screenshot for Case 3

4.3.4 Case 4: Expression (A ∧ B) ∨ (A ∧ C)∨ (B ∧ C); test set TTT from Figure 4.4

The expression in Case 4 includes multiple occurrences of conditions. Because the test set contains only one test case, there is no pair of test cases for any condition; thus, we estimated MC/DC as 0%. The Coco tool provides 0% for the same reason (Figure 4.14).

a	b	a	c	b	c	Decision	Description
TRUE	FALSE	TRUE	FALSE	TRUE	FALSE	FALSE	never executed
FALSE		TRUE	FALSE	TRUE	FALSE	FALSE	never executed
TRUE	FALSE	FALSE		TRUE	FALSE	FALSE	never executed
FALSE		FALSE		TRUE	FALSE	FALSE	never executed
TRUE	FALSE	TRUE	FALSE	FALSE		FALSE	never executed
FALSE		TRUE	FALSE	FALSE		FALSE	never executed
TRUE	FALSE	FALSE		FALSE		FALSE	never executed
FALSE		FALSE		FALSE		FALSE	never executed
TRUE	TRUE					TRUE	executed 1 time by 1 test
TRUE	FALSE	TRUE	FALSE	TRUE	TRUE	TRUE	never executed
FALSE		TRUE	FALSE	TRUE	TRUE	TRUE	never executed
TRUE	FALSE	FALSE		TRUE	TRUE	TRUE	never executed
FALSE		FALSE		TRUE	TRUE	TRUE	never executed
TRUE	FALSE	TRUE	TRUE			TRUE	never executed
FALSE		TRUE	TRUE			TRUE	never executed

Figure 4.14: Coco screenshot for Case 4

However, even if for only one test case, CodeCover provides some MC/DC coverage (see in Figure 4.15). The tool considers that this test case provides 50% for first A and 50% for first B, but does not cover other entities because of masking. Thus, two out of six occurrences have 50% coverage, which yields (2/6)*50 % = 16.7% (Figure 4.15). However, just one line of the test case should not show any MC/DC coverage because one line of test case can never verify all possible combination for any condition.

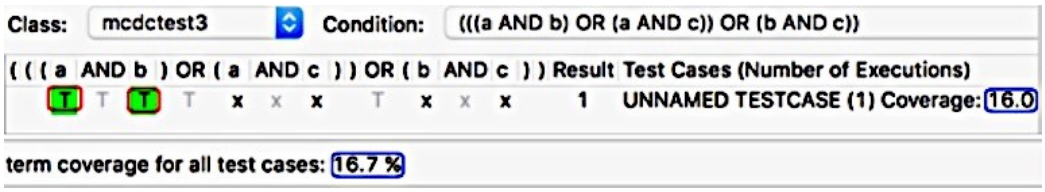

Figure 4.15: CodeCover screenshot for Case 4

Figure 4.16 shows how CTC++ evaluates the coverage of this logical expression.

Only one point is given for T value of the decision. The total number of conditions is considered to be six because multiple occurrences of the same condition are considered distinct entities. Therefore, the total coverage is (1/8) * 100% = 12.5%.

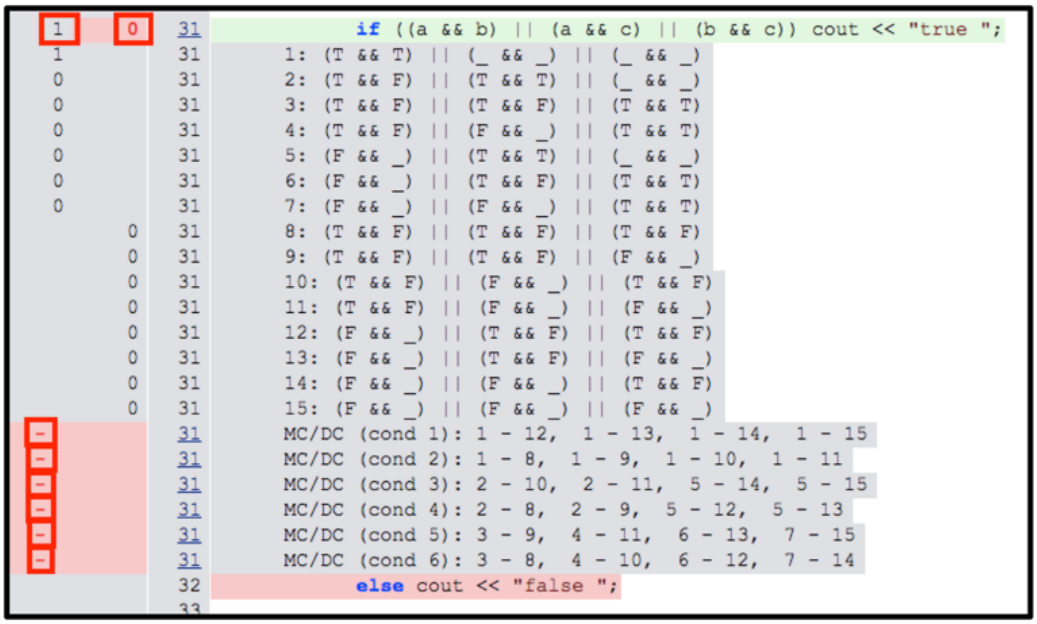

Figure 4.16: Testwell CTC++ screenshot for Case 4

Chapter 5

MCDC Application Development

The goal of the development process was to develop an MC/DC coverage measuring tool which calculates MC/DC coverage in the right way. This is very important because most of the current tools do not do this properly as discussed in the previous chapter 4. There is no other tool in the market which calculates the MC/DC coverage the way we do.

5.1 USE CASE DIAGRAM

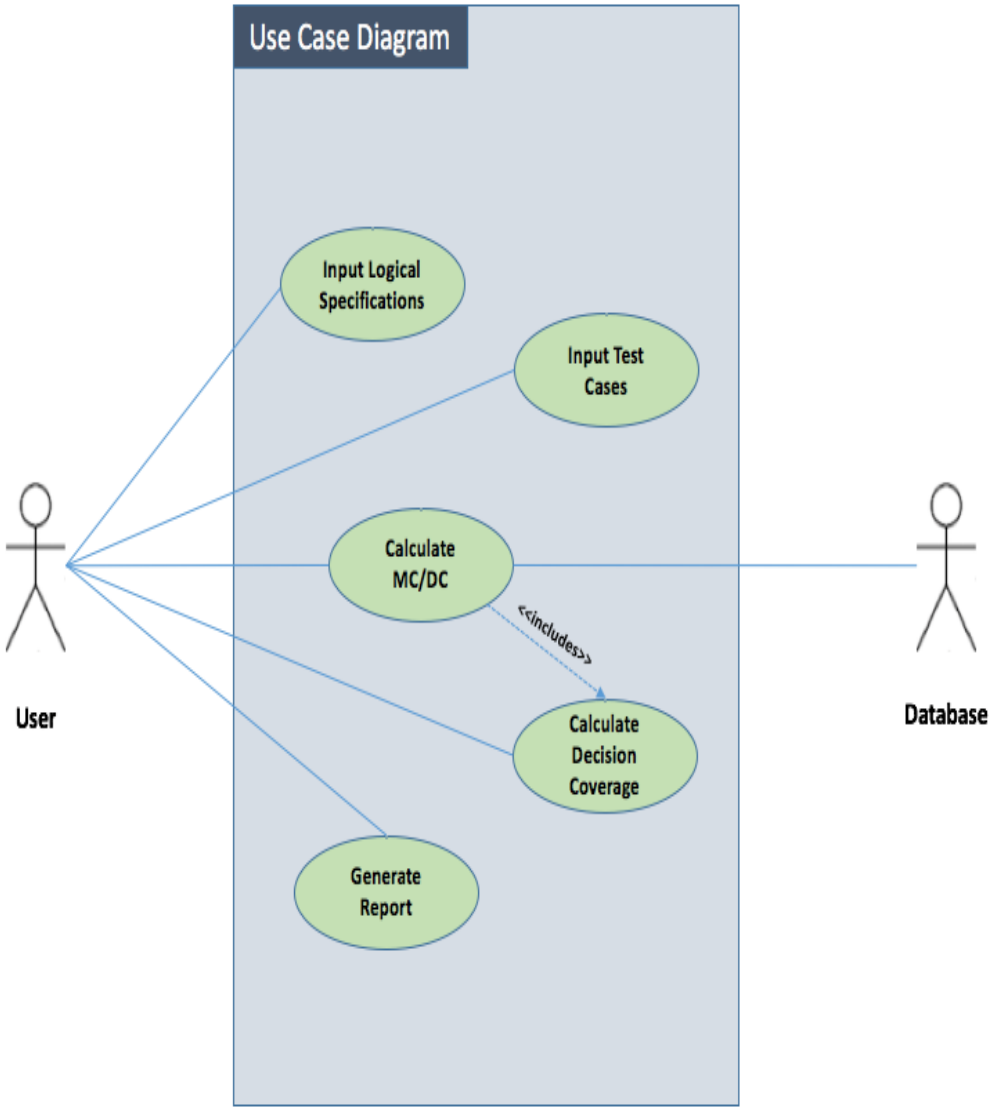

Figure 5.1: Use case diagram

In Figure 5.1, there are primarily two types of user in the application tool. One is the user who can access the input logical specification by selecting the decision file from the tool, the input text file by selecting the test case file from the tool, calculate MC/DC, calculate decision coverage and generate report of all the coverages. An-

other type of the user is the database in the system which allows the user to calculate and analyze the MC/DC coverage in the tool. Calculate MC/DC coverage use case includes Calculate Decision Coverage use case because when we calculate MC/DC coverage for a particular condition it calculates all possible decision values for that logical specification too.

5.2 Class Diagram

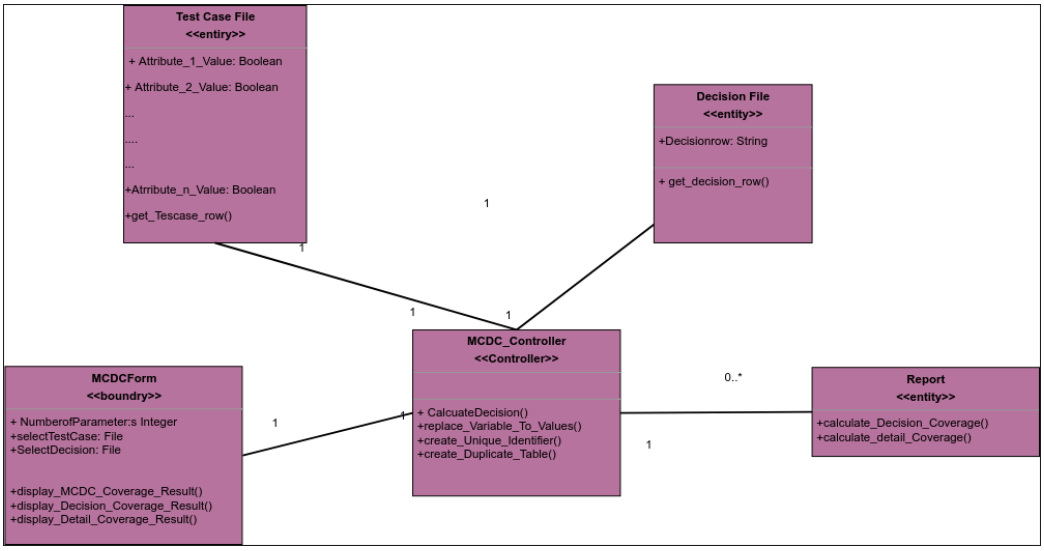

Figure 5.2: Class Diagram

In this development project agile development principles [18] are used so that unwanted documentation and any resulting complexity could be reduced. The main idea is to keep it simple, in a way that could be easily modifiable. That is the reason why the entire SRS document has been reduced to just a set of eleven User Stories cards. In addition to this, the entire project management and design steps have been

reduced to Product Backlog, Spring Backlog, Use Case diagram and Class Diagram (see Figure 5.2), respectively.

5.3 User Story Cards

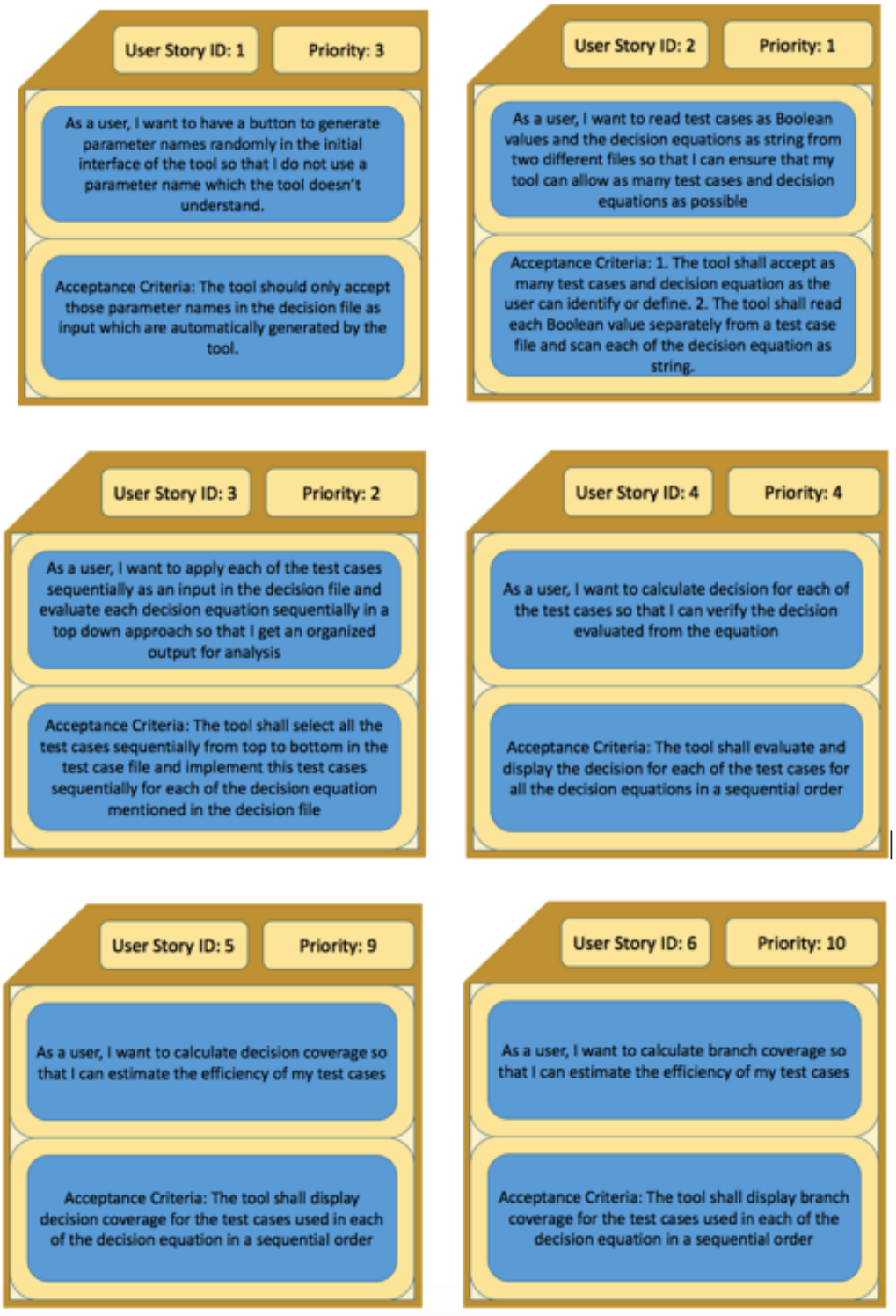

Figure 5.3: User Story

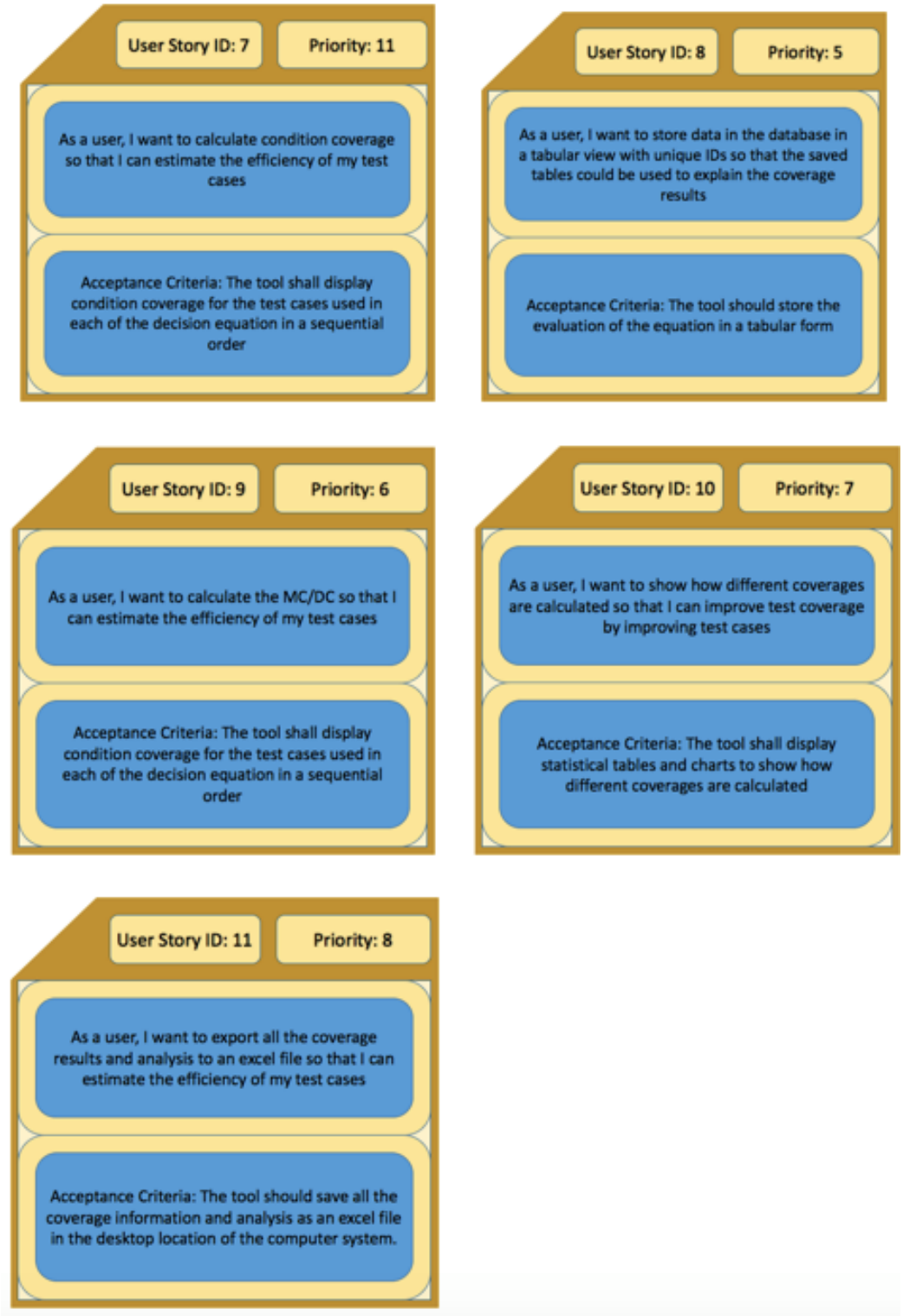

Figure 5.4: User Story

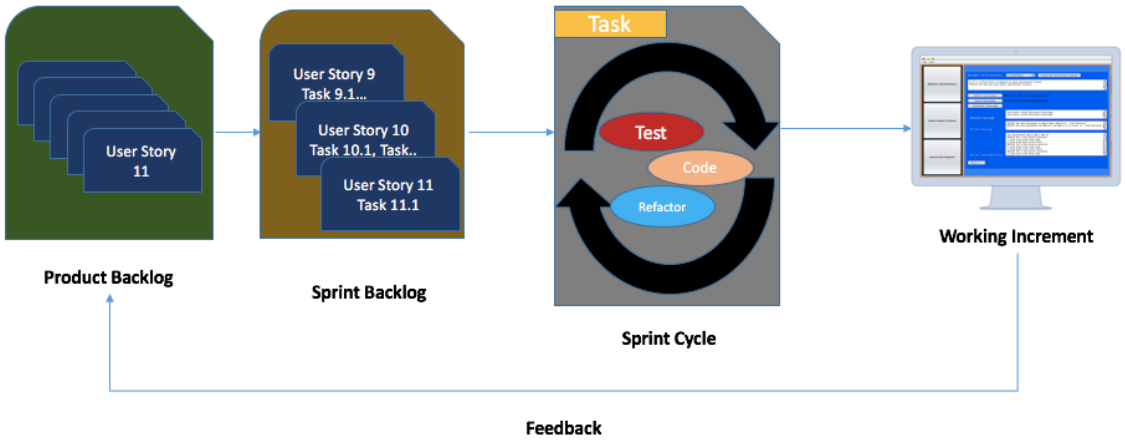

Figure 5.5: Scrum process for the project implementation

The following are the eleven user stories [18] identified by the user for the entire software application. User Story ID and the estimated size of each of the user story have been mentioned at the top of each User Story card (Figures 5.3, 5.4). Only functional requirements are considered for these user stories. Non-functional requirements are considered at a lower level during the implementation of the software application; however, those were not documented at this present moment, as our goal is to minimize unnecessary documentation, which is one of the key Agile principles.

5.4 SCRUM DEVELOPMENT CYCLE

The above Figure 5.5 explains how the User Stories were further fragmented and distributed among several sprints in the sprint backlog. Each of these user stories were further fragmented into several tasks in the sprint backlog. These tasks in each of the sprint produced a product after a process of code, test and re-factor, which was given to the professor for the acceptance level testing, in which many different feedback responses were collected. In the very next sprint, the required changes were made to those tasks.

The following is the Product Backlog [18], in which each of the User Stories have been mentioned, along with the theme, priority, progress status, criticality, size and sprint number (See Figure 5.6). All the User Stories were distributed and assigned to finish within 5 sprints. Where one user story from sprint 4 and all the user stories from sprint 5 has been identified with least priority and it is not expected to be a part of the thesis. However, these are additional functionality which are usually expected from all types of coverage measuring tools in the market but since the topic of the research is specific to MC/DC, we are not covering those at this moment.

PRODUCT BACKLOG

User Story ID	Associated Theme	As a/an	I want to...	So that...	Priority	Status	Criticality Low 3 Medium 2 High 1	Size XL L S	Sprint
1	Interface	User	have a button to generate parameter names randomly in the initial interface of the tool	I do not use a parameter name which the tool doesn't understand	3	In Process	2	L	1
2	User Input Type	User	read test cases as Boolean values and the decision equations as string from two different files	I can ensure that my tool can allow as many test cases and decision equations as possible	1	Complete	1	S	2
3	User Input Type	User	apply each of the test cases sequentially as an input in the decision file and evaluate each decision equation sequentially in a top down approach	I get an organized output for analysis	2	Complete	1	L	2
4	Backend Evaluation	User	calculate decision for each of the test cases	I can verify the decision evaluated from the equation	4	Complete	1	XL	3
5	Backend Evaluation	User	calculate decision coverage	I can estimate the efficiency of my test cases	9	Complete	3	S	3
6	Backend Evaluation	User	calculate branch coverage	I can estimate the efficiency of my test cases	10	Incomplete	3	L	5
7	Backend Evaluation	User	calculate condition coverage	I can estimate the efficiency of my test cases	11	Incomplete	3	L	5
8	Database	User	store data in the database in a tabular view with unique IDs	the saved tables could be used to explain the coverage results	5	Complete	1	XL	4
9	Backend Evaluation	User	calculate the MC/DC	I can estimate the efficiency of my test cases	6	Complete	1	XL	4
10	Interface	User	show how different coverages are calculated	I can improve test coverage by improving test cases	7	Complete	1	L	4
11	Interface	User	export all the coverage results and analysis to an excel file	I can estimate the efficiency of my test cases	8	Pending	2	L	4

Figure 5.6: Product Backlog

The following is the Sprint Backlog [18], where each of the User Stories have been further identified into several tasks (See in Figure 5.7). Each of the tasks are arranged sequentially:

SPRINT BACKLOG

Sprint	User Story ID	Task	Status	Owner	Estimated Hours	Actual Hours
1	1	1.1. Create NetBeans project with jframe	Complete	Gourav	2	1
		1.2. Create different panels and connect them with buttons.	Complete	Gourav	5	4
		1.3. Create a drop down button with number of conditions as option, a "Generate Parameter Names" button and a text area to show the results.	Complete	Gourav	3	2
2	2	2.1. Create test case file with all the test cases as Boolean input values.	Complete	Gourav	1	1
		2.2. Create decision file with all the decision equation. Equation should consist of both operators and operands.	Complete	Gourav	1	1
		2.3. Scan the input test cases to calculate total number of conditions and total number of test cases present in the file.	Complete	Gourav	5	6
	3	3.1. Substitute value of each of the test cases one by one in the decision in a top down approach.	Complete	Gourav	8	8
		3.2. Repeat steps in 3.1 for all the decision equation in the decision folder one by one in a top down approach.	Complete	Gourav	2	2
3	4	4.1. Evaluate decision value for all the decision equation after the conditions are substituted with the test case values.	Complete	Gourav	20	25
	5	5.1. Check decision coverage percentage for each of the decision equation individually.	Complete	Gourav	2	2
4	8	8.1. Set up SQLite Database and JDBC connection.	Complete	Gourav	5	5
		8.2. Create table for each decision equation and declare the datatypes of columns.	Complete	Gourav	5	5
		8.3. Generate unique IDs for each rows in the tables.	Complete	Gourav	2	3
		8.4. Add all the decision output generated for each test case as a column in the table.	Complete	Gourav	5	3
		8.5. Save each of these actual tables in the database for result analysis.	Complete	Gourav	4	4
	9	9.1. Create duplicate tables for all the tables created in User Story 8.	Complete	Gourav	5	5
		9.2. Compare each of the rows from actual table to all the rows in the duplicate tables to evaluate MC/DC coverage.	Complete	Gourav	15	20
		9.3. Calculate MC/DC.	Complete	Gourav	4	5
	10	10.1. Display each of the condition which shows MC/DC coverage with the contains of the rows which shows MC/DC behavior.	Complete	Gourav	2	1
		10.2. Display how many number of times each condition has been evaluated in the decision equation.	Complete	Gourav	3	3
		10.3. Display MCDC formula that is total number of times each condition has covered MC/DC divided by total number of occurrence of all conditions.	Complete	Gourav	4	4
	11	11.1. Exporting test results in excel file.	Pending	Gourav	3	N/A
5	6	6.1. Calculate branch coverage	Pending	Gourav	10	N/A
	7	7.1. Calculate condition coverage	Pending	Gourav	5	N/A

Figure 5.7: Sprint Blacklog

5.5 SCREENSHOT OF THE TOOL

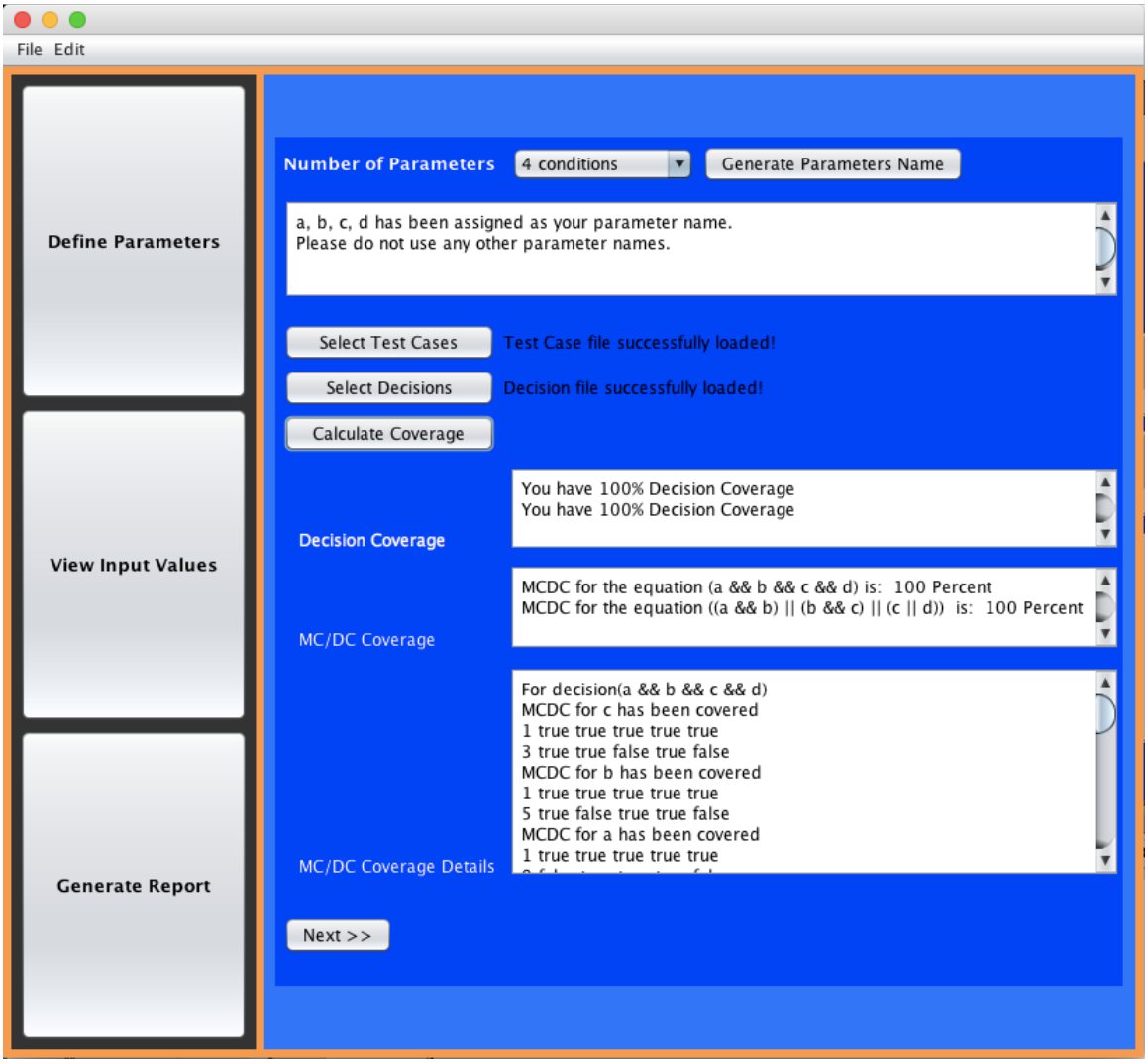

Figure 5.8: Tool Screenshot

The above screenshot in Figure 5.8 portrays the main page of the tool, which consists of several buttons, a drop down menu, text areas, and labels. This entire tool was developed using Java and Java Swing. At the backend, SQLite database was utilized to calculate MC/DC coverage and store the output table, which is used for comparing the results displayed in the MC/DC Coverage Details in the picture shown above.

The first thing the user is expected to do when using the tool is described as follows: select number of conditions in the decision equation. The user is expected to select the number of conditions from the drop down menu at the top of the above picture. Once the number of parameters is selected and the "Generate Parameters Name" button has been clicked, the tool will automatically generate variable names for those parameters, which the user should use in the input Decision File. The user must do this, otherwise, those parameters will not be detected by the tool.

Next, the user is expected to select the Test Case file and Decision File using select test cases and select decision buttons. Once those files are identified and selected in the software application, the user should click on "Calculate Coverage".

Once the "Calculate Coverage" button has been clicked, the user can see decision coverage in the tool‘'s decision coverage text area. This will display the decision coverage results for each of the individual decisions. Along with the decision coverage results, the user will also get to see MC/DC coverage for each of the decisions, as well as MC/DC coverage details for each of the decisions.

In the MC/DC coverage details text area of the software application, MC/DC coverage of each individual condition has been displayed along with the row numbers which are showing MC/DC coverage for it. Only those conditions are displayed here which shows MC/DC coverage. Also, there are situations where multiple test cases are showing MC/DC for a particular condition. Each of those situations is also uniquely shown in the same text area.

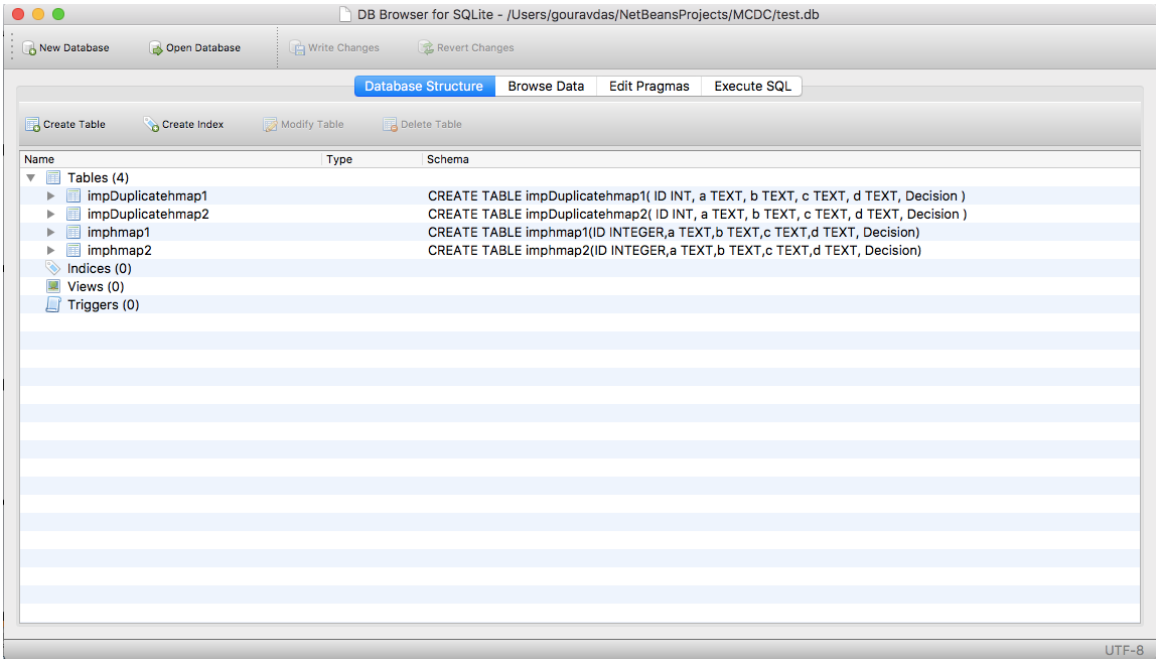

Figure 5.9: DB Browser for SQLite Database

SQLite Database has been used to store analyses for MC/DC coverage where each decision has two tables for an instant decision one has impmap1 and impduplicatemap1 as shown in Figure 5.9. Both of these tables are duplicate copies. The purpose of making it duplicate is to compare each row from a table to its duplicate table. In database table rows cannot be compared to the same table and other approaches will make the coding more complex. To avoid unnecessary complexity of code and to store MC/DC table for coverage analysis, the database has been used.

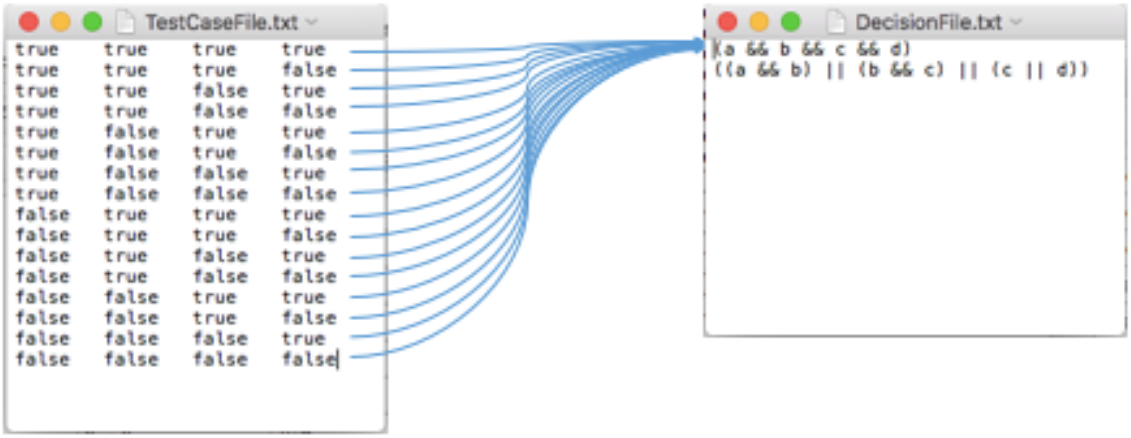

Figure 5.10: Test Case and Decision files

In the above Figure 5.10, each test cases from the TestCaseFile are sequentially evaluated in a top down approach for each of these decision equation mentioned in the DecisionFile. Then each of these decision results is stored in the Decision column of the database table against its test case.

Table: imphmap1

	ID	a	b	c	d	Decision
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	true	true	true	true	true
2	2	true	true	true	false	false
3	3	true	true	false	true	false
4	4	true	true	false	false	false
5	5	true	false	true	true	false
6	6	true	false	true	false	false
7	7	true	false	false	true	false
8	8	true	false	false	false	false
9	9	false	true	true	true	false
10	10	false	true	true	false	false
11	11	false	true	false	true	false
12	12	false	true	false	false	false
13	13	false	false	true	true	false
14	14	false	false	true	false	false

Figure 5.11: SQLite database table

For each test case row, unique row ID has been generated in the database which not only helps to uniquely identify each row but also it is used in the tool‘s output to explain which rows are showing MC/DC coverage for which conditions (See in Figure 5.21).

5.6 HIGH LEVEL CODING VIEW

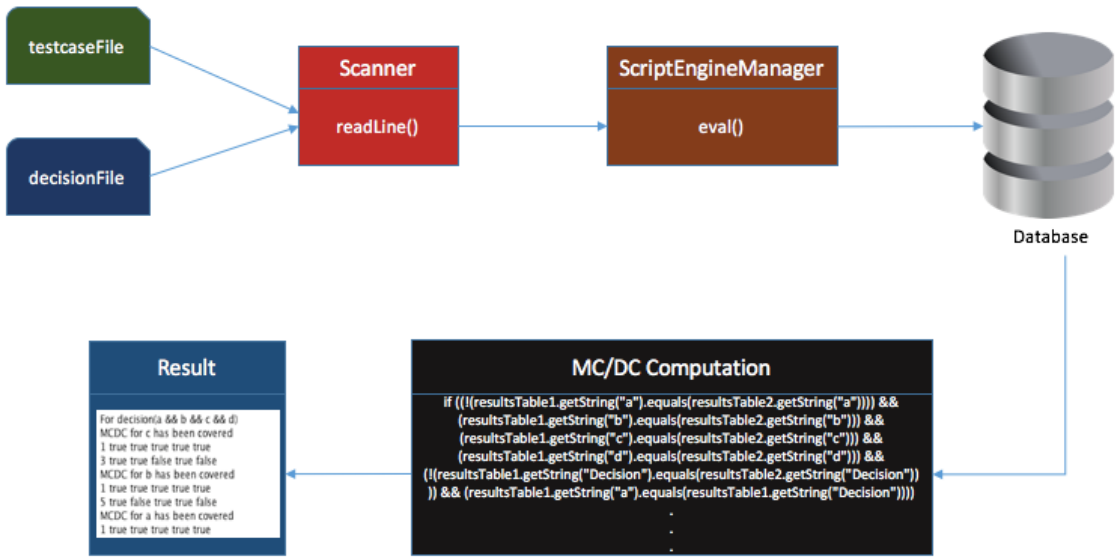

Figure 5.12: Coding details

The core portion of the code starts with two files chooser for both the decision file and test case file as shown in Figure 5.12. Test case file has all the test cases in terms of true and false values and on the other side, the decision file has all the decision equations in terms of the operator and operands (see Figure 5.10).

The scanner then reads these files line by line in a top down approach using while loop and decision.hasNext() and testcases.hasNext() methods. The first scanner reads the test case values word by word and then replaces the respective conditions which it belongs to in the decision string with these test case values. See the following code where orgString contains decision string from the decision file:

In the code below, str1, str2, str3, str4 contains values of all the four conditions from the test case in the test case file. At the end of the above code, we get the

```
if (orgString.contains("a")) {
                    str1 = String.valueOf(a);
                    orgString = orgString.replaceAll("a", str1);


                    if (orgString.contains("b")) {
                        str2 = String.valueOf(b);
                        orgString = orgString.replaceAll("b", str2);


                        if (orgString.contains("c")) {
                            str3 = String.valueOf(c);
                            orgString = orgString.replaceAll("c", str3);


                            if (orgString.contains("d")) {
                                str4 = String.valueOf(d);
                                orgString = orgString.replaceAll("d", str4);

                            }
                        }
                    }
                }
```

Listing 1: Test case values replacement for decision variables

final string in orgString which we then evaluate using the Script Engine Manager to produce the final true or false decision value.

Once we have the final decision value for each test case, we store this value next to the input value for that test case in the SQLite database table. For managing the database and to uniquely identify all the test case rows in the database table, another column ID has been created in the table to uniquely identify each test cases with its unique decision value.

After the database table is ready, we use Result Set object in JDBC to navigate through the tables. As we are only allowed to navigate in the forward direction in SQLite database we have created a duplicate table to compare each row of the table

with each other and to identify which row is showing the MC/DC coverage for which condition.

Finally, to calculate MC/DC coverage we first count the total number of conditions in a test case, then we use the following formula to get the final MC/DC coverage percentage.

MC/DC coverage = (Number of times condition A has occurred in the decision if covered MC/DC + Number of times condition B has occurred in the decision if covered MC/DC + Number of times condition C has occurred in the decision if covered MC/DC + Number of times condition D has occurred in the decision if covered MC/DC)/ Total number of times condition A, B, C, and D has occurred in the decision) * 100.

For instance, in an expression with three conditions (A $\wedge$ B) $\vee$ (A $\wedge$ C) $\vee$ (B $\wedge$ C), number of times A has occurred is 2, number of times B has occurred is 2, and number of times C has occurred is 2. If we insert TTF, FTF, and TFF as test cases then we get MC/DC coverage for condition B and C only. This gives us MC/DC coverage as follows:

MC/DC coverage = (Number of times condition A has occurred in the decision if covered MC/DC + Number of times condition B has occurred in the decision if covered MC/DC + Number of times condition C has occurred in the decision if covered MC/DC)/ Total number of times condition A, B, C, and D has occurred in the decision) * 100.

Which implies, MC/DC coverage = (0 + 2 + 2)/ 2+2+2 = 4/6 = 2/3 = 66.66%.

5.7 MC/DC Tool results:

Our MC/DC tool's Results			
Expression	Test set	MC/DC Level, %	
		Black-box, Manual (Expected)	Our Results (Actual)
A ∨ B	TF, FT, FF.	100	100
A ∧ B ∧ C	TTT, TTF, TFT, FTT.	100	100
A ∨ (B ∧ C)	TFT, FTT, FFT, FTF.	100	100
	TTF, TFT, FTF, TFF.	100	100
(A ∧ B) ∨ (A ∧ C) ∨ (B ∧ C)	TFT, FTT, FFT, TFF.	100	100
	TTF, FTT, FTF, TFF.	100	100

Figure 5.13: Table of MC/DC coverage result from our tool (100 % Coverage).

Our MC/DC tool's Results			
Expression	Test set	MC/DC Level, %	
		Black-box, Manual (Expected)	Our Results (Actual)
A ∨ B	TT	0	0
	FT	0	0
	TF, FF.	50	50
	FT, FF.	50	50
A ∧ B ∧ C	TTT	0	0
	FFF	0	0
	TTT, FTT.	33.33	33
	TTT, TFT.	33.33	33
	TTT, FTT, TFT.	66.66	66
	TTT, TFT, TTF.	66.66	66
A ∨ (B ∧ C)	TFF	0	0
	FTF	0	0
	TFT, FFT.	33.33	33
	TTF, TFF.	33.33	3
	TFT, FFT, FTT.	66.66	66
	FTT, FTF, FFT.	66.66	66
(A ∧ B) ∨ (A ∧ C) ∨ (B ∧ C)	TTT	0	0
	FFF	0	0
	TTF, FTF.	33.33	33
	FTT, FTF.	33.33	33
	TTF, FTF, TFF.	66.66	66
	FTT, FTF, TTF.	66.66	66

Figure 5.14: Table of MC/DC coverage result from our tool (LESS than 100 % Coverage).

After running our tool, we have verified that all the test cases are showing the result as expected. The results are shown in Figure 5.13 and Figure 5.14.

Chapter 6

Related Work

The literature review of MC/DC [5] found 54 research publications that discussed different MC/DC definitions. Only nine papers considered MC/DC as a black-box approach. These papers focus mainly on test generation and fault detection rather than measuring coverage as in our thesis.

However, there are other research papers on black-box requirements coverage criteria that refer to MC/DC by different names. Thus, a requirements coverage metric referred to as the Unique First Cause Coverage was adapted from the MC/DC criterion [19], [20], [21]. This criterion extends the constraints for MC/DC using temporal operators of the Linear-Time Temporal Logic.

Reinforced Condition/Decision Coverage (RC/DC), another extension of MC/DC for requirements coverage, has also been suggested [4, 22]. RC/DC requires that test cases should maintain a constant value for a decision when varying a condition. This allows testing safety-important situations when a false actuation of a system is possible.

The Full Predicate Coverage criterion, similar to MC/DC, has been developed for specification-based testing [23]. Several other criteria, including Restricted Active Clause Coverage (RACC) and Correlated Active Clause Coverage (CACC) [24], are also suitable for requirements coverage and are associated with MC/DC. Thus,

unique-cause MC/DC corresponds to RACC, and masking MC/DC corresponds to CACC.

Model-based MC/DC test case generation from different models of software specifications is considered in many research papers [25, 26, 27, 28, 29, 30]. The main content of these investigations addresses how to achieve 100% MC/DC for formal logical requirements. Only a few papers suggest methodologies to evaluate MC/DC coverage of the models. Thus, a dynamic backward slicing technique is used in [30] to combine results from code coverage with results from requirements coverage. The requirements are presented as assertions over program variables. Different types of code coverage are first evaluated and then mapped back to the model. However, no clear rules on how to measure MC/DC model coverage are presented.

The experimental results of evaluation of MD/DC coverage levels of pair-wise test cases for logical specifications were provided in [31]. The evaluations were done for separate logical specifications in different modes and sizes. The MC/DC coverage for pair-wise test cases was compared to the MC/DC coverage for random test cases of the same size. Another experimental evaluation of the ability of random testing to provide coverage of logical specifications according to MC/DC was presented in [32].

Chapter 7

Future Work

- To develop this tool further to calculate MC/DC for any number of conditions in requirements specifications.
- Developing a tool to convert all the requirements of a specific format to the logical representation.
- Use this developed tool as a reusable component or system to calculate MC/DC during different phases of software development lifecycle. This will significantly help to detect bugs at the very early stages of the development lifecycle.
- This developed MC/DC coverage tool could not only be used with the logical specification but also with another form of design specifications like State Machine Diagram.
- This developed tool could be also used in the code as a white box testing approach if we can identify and separate out all the decision equation from the code.
- We can not only use this tool for generating MC/DC coverage but also generate test cases for MC/DC coverage by first generating all the possible combination of test cases with the given number of conditions. Then insert all those possible combinations of conditions into my MC/DC tool to generate test cases which are showing MC/DC coverage. Then we can put all these test cases which are showing MC/DC into a database table where we can eliminate duplicate rows. Then finally we can show this table to the user as a suggested list of test cases which together shows 100% MC/DC coverage. Now, it is up to the user what test cases they want to use from it.

We believe that the investigation and comparison of the fault detection ability of white-box and black-box MC/DC is an important direction for future work.

Chapter 8

Conclusion

In this thesis, we considered a well-known MC/DC approach as a black-box specification or requirement-based testing technique. We concentrated on the estimation of the MC/DC level for logical specifications/requirements, especially when this coverage is less than 100%. The results of our investigation allowed us to reach the following conclusions:

- MC/DC can be used both for white-box and black-box testing, but the application of MC/DC as a black-box technique has not been studied sufficiently.
- It often is simpler and more natural to use MC/DC for black-box rather than white-box testing. Many factors should be considered differently for these two modes.
- No common approach exists to calculate MC/DC.
- Different code coverage tools are based on different coverage principles, and their MC/DC results vary significantly.
- These tools are not suitable for MC/DC estimation of logical requirements.
- Decision coverage is calculated by default even if one condition satisfies MC/DC coverage criteria. No need to calculate it separately.
- The best approach to the MC/DC estimation of logical requirements is to estimate the coverage for each separate condition in a decision, after which the results can be combined.
- The test cases produced by calculating MC/DC on the requirement specification could be even used in the codes.

- New tool has been developed which is suitable to calculate MC/DC coverage under this research project.
- This tool is capable of calculating decision coverage for all the logical expressions
- This tool can explain which test cases shows MC/DC coverage for each individual condition in an expression if at all it has MC/DC coverage.
- This tool produces database table which could be used to analyze how the MC/DC coverage has been calculated.
- The tool developed for this project works exactly the way we have described in the thesis.
- At this moment this tool works as a prototype for only a fixed number of conditions at a particular time.
- This tool can be even used to generate test cases if all possible combination of test cases are mentioned in the test case file as an input.

BIBLIOGRAPHY

[1] J. J. Chilenski, "An investigation of three forms of the modified condition decision coverage (mcdc) criterion," BOEING COMMERCIAL AIRPLANE CO SEATTLE WA, Tech. Rep., 2001.

[2] L. A. Johnson *et al.*, "Do-178b, software considerations in airborne systems and equipment certification," *Crosstalk, October*, vol. 199, 1998.

[3] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[4] S. A. Vilkomir and J. P. Bowen, "From mc/dc to rc/dc: Formalization and analysis of control-flow testing criteria," in *Formal methods and testing*. Springer, 2008, pp. 240–270.

[5] T. K. Paul and M. F. Lau, "A systematic literature review on modified condition and decision coverage," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. ACM, 2014, pp. 1301–1308.

[6] S. Vilkomir, J. Baptista, and G. Das, "Using mc/dc as a black-box testing technique," in *Software Technology Conference (STC), 2017 IEEE 28th Annual*. IEEE, 2017, pp. 1–7.

[7] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[8] I. S. Committee *et al.*, "Ieee std 610.12-1990 ieee standard glossary of software engineering terminology," *online] http://st-dards. ieee. org/reading/ieee/stdpublic/description/se/610.12-1990 desc. html*, 1990.

[9] H. Zhu, P. A. Hall, and J. H. May, "Software unit test coverage and adequacy," *Acm computing surveys (csur)*, vol. 29, no. 4, pp. 366–427, 1997.

[10] a. K. P. Alagar, "Specifications of software system," 2001.

[11] Mathswork, "Simulink verification and validation." [Online]. Available: https://www.mathworks.com/products/simverification.html

[12] MathsWorks, "Simulink design verifier." [Online]. Available: https://www.mathworks.com/products/sldesignverifier.html

[13] "Modified condition and decision coverage (mcdc) definitions in simulink verification and validation." [Online]. Available: https://www.mathworks.com/help/slvnv/ug/modified-condition-and-decision-coverage-definitions-in-simulink-verification-and-validation.html

[14] MathsWorks, "Modified condition and decision coverage in simulink design verifier." [Online]. Available: https://www.mathworks.com/help/slvnv/ug/modified-condition-and-decision-coverage-in-simulink-design-verifier.html

[15] froglogic, ""coco code coverage tool, froglogic gmbh." [Online]. Available: https://www.froglogic.com/coco

[16] codecover, "Codecover." [Online]. Available: http://codecover.org

[17] V. T. GmbH, "Testwell ctc++ tool." [Online]. Available: http://www.verifysoft.com/en_ctcpp.html

[18] R. Black, G. Coleman, M. Walsh, B. Cornanguer, I. Forgacs, K. Kakkonen, J. Sabak, and R. Black, "Agile testing foundations: An istqb foundation level agile tester guide." BCS, 2017.

[19] M. W. Whalen, A. Rajan, M. P. Heimdahl, and S. P. Miller, "Coverage metrics for requirements-based testing," in *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006, pp. 25–36.

[20] M. Staats, M. W. Whalen, M. P. Heindahl, and A. Rajan, "Coverage metrics for requirements-based testing: Evaluation of effectiveness," 2010.

[21] A. Rajan, *Coverage metrics for requirements-based testing*. University of Minnesota, 2009.

[22] S. A. Vilkomir and J. P. Bowen, "Reinforced condition/decision coverage (rc/dc): A new criterion for software testing," in *International Conference of B and Z Users*. Springer, 2002, pp. 291–308.

[23] A. J. Offutt, Y. Xiong, and S. Liu, "Criteria for generating specification-based tests," in *Engineering of Complex Computer Systems, 1999. ICECCS'99. Fifth IEEE International Conference on*. IEEE, 1999, pp. 119–129.

[24] P. Ammann, J. Offutt, and H. Huang, "Coverage criteria for logical expressions," in *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on.* IEEE, 2003, pp. 99–107.

[25] A. C. de Melo, C. S. Păsăreanu, and S. Hanazumi, "Towards mc/dc coverage of properties specification patterns," in *International Colloquium on Theoretical Aspects of Computing.* Springer, 2016, pp. 158–175.

[26] S. Weißleder, "Test models and coverage criteria for automatic model-based test generation with uml state machines," Ph.D. dissertation, Humboldt University of Berlin, 2010.

[27] R. Bloem, K. Greimel, R. Könighofer, and F. Röck, "Model-based mcdc testing of complex decisions for the java card applet firewall," in *VALID 2013, Fifth Int. Conf. Adv. Syst. Test. Valid. Lifecycle*, 2013, pp. 1–6.

[28] A. Pretschner, "Compositional generation of mc/dc integration test suites," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 1–10, 2003.

[29] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," in *Engineering of Computer Based Systems, 2001. ECBS 2001. Proceedings. Eighth Annual IEEE International Conference and Workshop on the.* IEEE, 2001, pp. 83–91.

[30] A. Murugesan, M. W. Whalen, N. Rungta, O. Tkachuk, S. Person, M. P. Heimdahl, and D. You, "Are we there yet? determining the adequacy of formalized requirements and test suites," in *NASA Formal Methods Symposium.* Springer, 2015, pp. 279–294.

[31] S. Vilkomir and D. Anderson, "Relationship between pair-wise and mc/dc testing: Initial experimental results," in *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on.* IEEE, 2015, pp. 1–4.

[32] S. Vilkomir, A. Alluri, D. R. Kuhn, and R. N. Kacker, "Combinatorial and mc/dc coverage levels of random testing," in *Software Quality, Reliability and Security Companion (QRS-C), 2017 IEEE International Conference on.* IEEE, 2017, pp. 61–68.